Programming Language: Numera

Team members:

Team member	**Role**	**UNI**
Meng Gao	Manager, Language Guru	mg4774
Yixuan Li	System Architect, Compiler Architect	yl3803

Weekly meeting time: Monday 8 - 9 pm

Introduction:

Numera is a versatile and intuitive programming language designed for mathematical computation. It supports a range of operations from basic arithmetic to advanced mathematical functions, making it ideal for both simple calculations and more complex numerical analysis. Overall, Numera is perfect for applications that require flexibility in data representation and mathematical processing.

Motivation:

Many existing programming languages either excel in mathematical precision or user-friendly syntax but rarely both. Numera seeks to bridge this gap by offering a language that is both simple to use and powerful enough to handle complex numerical tasks. Numera's flexibility in data representation and mathematical processing provides possibilities for a wide range of applications. We hope our users will focus on solving problems rather than wrestling with language complexities. In short, we want to create a programming language that merges simplicity with functionality.

Creativity/Novelty:

While several features of Numera can be found in existing languages (Python, MATLAB, Julia), its novelty lies in its dedication to simplicity, intuitive design for mathematical computation, and unique features like unit conversion. The combination of these traits could make it an attractive language for math students and professionals who need a quick and flexible way to perform and automate calculations without diving into the complexities of more general-purpose programming languages.

Language and Compiler Description:

- Arithmetic & math functions: perform operations with basic arithmetic symbols (`+,-,*,/,%`) and advanced functions like `sqrt()`, `log()`, `sin()`, and `cos()`.
- Equality & inequality comparisons: utilize operators like `==`, `!=`, `<`, `>`, `<=`, and `>=` for precise comparisons and conditional logic.
- Check brackets: `“()”  “[]”  “.”  “;”`.
- Loop constructs: implement iterative logic with `while` loops to handle repetitive tasks.
- Define `if` condition, and provide `and,or` condition. (e.g. `if a and b then, if a == b and a > c`).
- Unique variable check: each variable is unique, else report error.
- Receive user input in the format of `x = in();` if user input 6, x = 6.
- Print: use `print()` for output.
- Define unit conversation by `M(),KM(),CM()`. (e.g. `x = 1000cm M(x) x = 100m`)
- Define functions in the format of `procedure [func_name] ( arg1, arg2, … )`
- Report syntax or logic errors. `“Error: missing ;” “Error: Can’t divide 0 ” “Error: missing then”`

Example Simple Program Written in Numera

```
# add, subtract, multiply, divide and mathematical functions
var a = (10 + 20) * 2;
print(a); # output is 60;

result = sqrt(25) + sin(0) - log(10);
print(result);

max(); ====> max(10,20); return 20
min(); ====> min(10,20); return 10

# equality & inequality
var x = 10;
var y = 20;
if x < y then:
      print("x is less than y");
else:
      print("x is not less than y");
end

# while loop example
count = 0
```

```
while not count < 5 do:
    print(count)
    count = count + 1;
end

# main function & create custom function
procedure main is # main function start
     procedure MY_FUNCTION (arg1) is # new function created
          var a;
          if r > 9 then
               r = in();
          end
     end                  # new function end
     var x;               # global variable
begin                     # main function start
     x = 10;
     var y = A(x);
     print(y);
end # main function end

# error report
if x == 5   # program exit and report “Error: missing then”
     out(x);
end

while x > 3 do
     out(1);
     # program exit and report “Error: missing end”

var a = 0;
var b = 10/a; # report “Error: Divide 0”

var a = 10 + 20 + (20 + 40)); # report “Error: Missing (“
var a = 10 * 2 /; # report “ Error: syntax errors”
procedure A ( r ) # report “ Error: missing is”
          var a;
          if r > 9 then
               r = in();
```

```
           end
end
```

All the terms are case sensitive and should follow our rule, otherwise we will report errors.
Error cases: `Begin End For While whIle eND proceDure`

# unit conversion provided:

```
int x = 100m   M(x) x = 1m CM(x) x = 10000cm  KM(x) x = 0.1km
```

# syntactic logic definition

```
<procedure> ::= procedure main is <decl-seq> begin <stmt-seq> end
              | procedure main is begin <stmt-seq> end
<decl-seq> ::= <decl> | <decl><decl-seq>
             | <function> | <function><decl-seq>
<stmt-seq> ::= <stmt> | <stmt><stmt-seq>
<decl> ::= <decl-var>
<function> ::= procedure ID <parameters> is <stmt-seg> end
<parameters> ::= id
<decl-var> ::= var id
<stmt> ::= <assign> | <if> | <loop> | <print> | <decl> | <call>
<call> ::= begin ID(<parameters>)
<assign> ::= id = <expr>
<print> ::= print(<expr>)
<if> ::= if <cond> then <stmt-seq> end
       | if <cond> then <stmt-seq> else <stmt-seq> end
<loop> ::= while <cond> do <stmt-seq> end
<cond> ::= <cmpr> | not <cond> | <cmpr> or <cond> | <cmpr> and <cond>
<cmpr> ::= <expr> == <expr> | <expr> != <expr> | <expr> < <expr>
         | <expr> > <expr> | <expr> <= <expr> | <expr> >= <expr>
<expr> ::= <term> | <term> + <expr> | <term> - <expr>
<term> ::= <factor> | <factor> * <term> | <factor> / <term>
<factor> ::= id | const | ( <expr> ) | in ( )
```